Randal L. Schwartz
Stonehenge Consulting Services, Inc.
+1 (503) 777 0095
http://www.stonehenge.com/merlyn/

Introduction to CGI::Prototype

Randal L. Schwartz
www.stonehenge.com/merlyn/
version 1.3 at 3 Mar 05

Overview

- *Why CGI::Prototype?*
- *The basic architecture*
- *A sample application*
- *CGI::Prototype::Hidden*

Why CGI::Prototype?

- *Web apps are backwards*
- *Ad hoc solutions don't scale*
- *Existing CPAN solutions weren't enough*
- *I had a client that needed a solution "yesterday" "and make it flexible!"*
- *Together with Class::DBI and Template Toolkit, I wanted to establish a "best of breed" trifecta*

Backwards web apps?

- *In traditional GUIs, the program drives the display, and therefore the user*
- *In a web app, the user drives the display, and therefore the app!*
- *Every web app must figure out how to manage control state between hits*
- *The logic is common, but the specifics are particular to an app*

The main flow

- *A web app needs to figure out initially what state a particular hit is in*
- *Based on that decision, some code must run to deal with the incoming data*
- *Based on those results, more code must run to generate some browser reponse*
- *Errors happen*
- *At bad times*

Ad hoc code

- *For a tiny web app, we end up writing the same thing:*

  ```
  if (param) {...response...} else {... show form ...}
  ```

- *The "param" here "dispatches" to one of two "pages", which have the responsibility to "respond" to the params and "render" a result*
- *But what if there's a user error?*
- *Now we need to show the form again!*

Separation of concerns

- *Let's separate the three parts*
  - *Dispatcher: decide which responder*
  - *Responder: analyze params, and decide which renderer*
  - *Renderer: generate the response*
- *And to further use best of breed, let's drive TT for most of the renderer, and use CGI.pm for forms and params*

Using objects

- *Let's also use objects—lots of them!*
- *Objects will allow similar behaviors to be captured, and distinct behaviors to be encoded or data-driven*
- *In fact, let's use Class::Prototyped for easy accessors and maximum flexibility*
- *And Class::DBI objects for data*

Separation of MVC

- *The buzzword is Model-View-Controller*
- *Class::DBI provides Model (data)*
- *Template Toolkit provides View (rendering back to the browser)*
- *All we need is a good reusable abstract Controller that we can subclass*
- *Enter CGI::Prototype*

The Basic Architecture

- *The controller is about as minimal as you can get*
- *Call "dispatch", which returns a "respond" object*
- *Call the "respond" object which returns a "render" object, containing a template*
- *Process "render" Template with TT*
- *Return the template output to STDOUT*

Hierarchy of classes

- *Create an application class, which inherits from CGI::Prototype*
- *Create render and respond objects, which inherit from the application class*
- *Create templates referenced by the respond objects*
- *It's simpler than the description, actually*

A sample application

- *The trivial sample application relies on all the default values:*

```
#!/usr/bin/perl
use base CGI::Prototype;
__PACKAGE__->activate; # main->activate;
```

- *This application says (using CGI.pm) "This page intentionally left blank"*
- *To override that, define a template:*

```
sub template { 'input.tt' }
```

Looking under the hood

- *The core of what ->activate is doing:*

```
my $self = shift; # $self is the app object
my $this_page = $self->dispatch;
my $next_page = $this_page->respond;
$next_page->render;
```

- *The default dispatch returns $self*
- *The default respond returns $self*
- *render calls $self->template, which returns the "blank" template, or our overridden filename*

Errors

- *That's all wrapped in an eval block*
- *Any exception calls $self->error($@)*
- *The default error handler dumps the error with a text/plain header*
- *You can override it to do what you want, of course*

The classic two-pass app

- *Let's do the typical "form—response"*
- *First, let's not use main:*

```
BEGIN {
  package MyApp;
  use base CGI::Prototype;
} # simulates "use MyApp" within this file
MyApp->activate;
```

- *Create our first pass:*

```
BEGIN {
  package MyApp::One;
  use base MyApp;
  sub template { 'the_form.tt' }
}
```

Second pass

- *We'll dispatch to the second pass if any params are seen*
- *Second pass needs to respond to params*
- *Since we can do that within the template, no code to write again!*

```
BEGIN {
  package MyApp::Two;
  use base MyApp;
  sub template { 'the_response.tt' }
}
```

Dispatcher

- *We'll need to dispatch to the first page if no params, and second if some, so let's add a dispatch:*

```
sub MyApp::dispatch {
  my $self = shift;
  return $self->param ? 'MyApp::Two' : 'MyApp::One';
}
```

- *Where is "param"? It's provided by the base class, just like ->CGI to get at the CGI.pm object*

the_form.tt

- *Templates are called with "self" as $self*
- *So self.CGI is the CGI object*
- *Thus we can call back easily:*

```
[% self.CGI.header %]<html><head></head><body>
<h1>Welcome!</h1>
Please enter your first and last name
[% self.CGI.start_form; self.CGI.textfield('first');
   self.CGI.textfield('last'); self.CGI.submit;
   self.CGI.end_form %]
</body></html>
```

- *Callbacks are also used for data fetch*

the_response.tt

- *A simple response based on the incoming parameters:*

```
[% self.CGI.header %]
<html><head></head><body>
<h1>Welcome!</h1>
Greetings, [% self.param('first') | html %]
[% self.param('second') | html %]!
</body></html>
```

- *Once again, a fully working application!*
- *View code is in TT, controller in Perl*
- *We can hand off the TT to a designer*

Dealing with problems

- *If there are params, but they aren't complete, we should render the form again instead of the response*
- *The dispatcher can check this, but...*
- *Better to check it in MyApp::Two*

Just override respond

- *We'll just override respond from the app-wide default of "return $self" to something that makes a choice:*

```
sub MyApp::Two::respond {
  my $self = shift;
  unless ($self->param('first') =~ /\S/ and
          $self->param('last') =~ /\S/) {
    return 'MyApp::One'; # back to the form
  }
  return $self;
}
```

- *So we think of these pages as "states"*

A better way to do that

- *We could have gone the other way:*

```
sub MyApp::dispatch { return 'MyApp::One' }
sub MyApp::One::respond {
  my $self = shift;
  if ($self->param('first') =~ /\S/ and
      $self->param('last') =~ /\S/) {
    return 'MyApp::Two'; # good params, move on
  }
  return $self; # bad params? stay here
}
```

- *This gives us the "stay here until you get it right" model*

Multi-page apps

- *If an app has more than two pages, the rules get more complicated*
- *Better to have some sort of "app state"*
- *The dispatcher can then use the correct ->respond based on the state*
- *The ->respond then decides the new state by selecting a page to ->render*
- *"stay on this page" for errors is easy*

Determining state in dispatch

- *How does the dispatcher know the page that most recently executed ->render?*
- *Could do it with a hidden field in a form*
  - *Hint: CGI::Prototype::Hidden*
- *Could do it with a mangled-URL*
- *Could do it with a cookie*
- *Could mix any of those with some server-side session database*

More hooks than a pirate

- *Lots of places to customize*
- *MUMBLE_enter: before calling*
- *MUMBLE_leave: after calling*
- *Example: app_enter called before anything else*
- *app_leave called after everything else*
- *By default, hooks do nothing*

Page “control”

- *A page has “control” while it is being either a responder or renderer or both*
- *So $page->control_enter is called just before a page gets control*
- *And $page->control_leave is called just after a page loses control*
- *Good place to put per-page setup and teardowns*

Page render or respond

- *Similarly, hooks are provided for respond_enter, respond_leave, render_enter, render_leave*
- *Called against the page that will do the respond or render*
- *Example: put code in render_enter hooks to preload sticky fields for CGI form generation*

Activate, fully rendered

- *In all of its glory:*

```
sub activate {
  my $self = shift;
  eval {
    $self->app_enter;
    my $this_page = $self->dispatch;     ### DISPATCH ###
    $this_page->control_enter; $this_page->respond_enter;
    my $next_page = $this_page->respond; ### RESPOND ###
    $this_page->respond_leave;
    if ($this_page ne $next_page) {
      $this_page->control_leave; $next_page->control_enter;
    }
    $next_page->render_enter;
    $next_page->render;                  ### RENDER ###
    $next_page->render_leave; $next_page->control_leave;
    $self->app_leave;
  };
  $self->error($@) if $@;
}
```

Rendering, in detail

- *“render” defaults to pushing $self->template through a TT engine, passing { self => $self } as the global vars*
- *Engine is $self->engine*
- *The results are displayed by passing them to $self->output (default STDOUT)*
- *override “output” for testing*

The TT engine

- *The default $self->engine creates a Template->new object with no parameters*
- *The engine is "autoloaded", so in mod_perl environments, it persists*
- *You can override ->engine to provide additional configuration controls*
- *Future CGI::P may have more hooks*

CGI::Prototype::Hidden

- *A subclass of CGI::Prototype*
- *Allows the "state" to be easily maintained in a hidden field*
- *Correlates the page classnames with the corresponding template names*
- *Classes lazy-loaded for good CGI speed*
- *Establishes "wrapper" template*
- *Easy framework for multi-page apps*

Defaults everywhere

- *config_state_param, default "_state": name of the param you have to put in every form as a hidden field so that the response is appropriately dispatched*
- *config_class_prefix, default "My::App": establishes My::App as the parent of My::App::This and My::App::That pages*

Initial page

- *config_default_page, default "welcome": the name of the class (and page) if no other page matches*
- *Thus, you will go to welcome.pm and welcome.tt on your first hit*

The wrapper

- *config_wrapper, default "My/App/WRAPPER.tt": default wrapper template*
- *Called to process each page*
- *Should call [% PROCESS $template %]*
- *Good place to define common macros or blocks, and put headers and footers*

The enhanced basics

- *dispatch looks for $self->param("_state")*
- *If not found, uses "welcome"*
- *Hands off response to My/App/welcome.pm*
- *respond returns render class (or $self)*
- *Default template for that class is My/App/welcome.tt*

The minimum app

- *You must provide “welcome.pm”, “welcome.tt”, and “WRAPPER.tt”,“My/App.pm” at least*
- *The TT search path is set to the @INC path (cheap hack), so your pm and tt files are in the same directory*
- *Every additional page: $x.pm, $x.tt, where $x comes from param(‘_state’)*

The hidden field

- *Every form must include the state param*

```
[% self.CGI.start_form %]
[% self.CGI.hidden(self.config_state_param) %]
... rest of form ...
[% self.CGI.submit; self.CGI.end_form %]
```

- *Without this, the dispatch gets lost*
- *Good idea to put this in WRAPPER as a macro or block*
- *If you generate URLs, also include the state param*

More things to override

- *render_enter_per_page: additional hook for per-page render items: use this instead of defining render_enter*
- *respond_per_app: return a page, or undef (defaults to calling respond_per_page)*
- *respond_per_page: return a page for render (defaults to $self)*

Callbacks

- *If the foo.tt needs some data, it can call self.some_method(arg1, arg2)*
- *Define this in foo.pm*
- *When some callback is needed by more than one page, move it into the App.pm*
- *Thus, the TT file never needs to do heavy lifting for data: push that to Perl code*

Data push

- *Instead of callbacks, use data push*
- *In render_enter(_per_page), add a slot:*

```
sub render_enter_per_page {
  my $self = shift;
  my @data = map { int rand 10 } 1 .. 10;
  $self->reflect->addSlot(data => \@data);
}
```

- *Now you can access this in TT:*

```
The lotto numbers are [% self.data.join(", ") %]
```

- *Be sure to clean up in render_leave*
- *Good place to fetch Class::DBI stuff*

App-wide notes

- *Clear a slot in the app during app_enter*

```
__PACKAGE__->reflect->addSlot(errors => []);
sub app_enter { shift->errors([]) } # reset
sub add_errors { push @{shift->errors}, @_ }
```

Now everyone can add a note:

```
unless ($self->param("first") {
  $self->add_errors("You forgot your first name");
  return $self; # stay on this page
}
```

- *Access this in the template:*

```
[% FOR e = self.errors;
IF loop.first; "<h2>Errors:</h2><ul>"; END;
"<li>"; e | html;
IF loop.last; "</ul>"; END; END; %]
```

Advantages of paged-based apps

- *You design a page as a TT file*
- *You put the response in ->respond*
- *That response decides "stay here" (for errors) or "go to next state"*
- *You add callbacks for heavy data*
- *It looks like you're calling the user, instead of the other way around*

Summary

- *CGI::Prototype—the archetypal web application*
- *CGI::Prototype::Hidden—a great base class for multi-page web applications*
- *Web applications—easier to write when you have the right tools*
- *More stuff on the way! Stay tuned!*